Transactions

Model:

Begin xact

Sql-1
Sql-2
.
.
.
Sql-n
commit or abort

Concurrency control (Isolation)
Crash recovery (Atomic, Durable)

Example: move $100 from acct-A to acct-B

Atomic: all or nothing
Durable: once done, it stays done
Isolation: produces the "right" answer in a concurrent world
Consistent: acct cannot go below zero

Consistent – deals with integrity constraints, which we are not going to talk about.

Concurrency control first:

Consider:

Bill	shoe	10K
Sam	shoe	15K
George	toy	12K
Hugh	toy	8K
Fred	shoe	14K

T1: Give a 10% raise to everybody in the shoe dept.
T2: Move George and Hugh into the shoe dept

Easy to create a parallel schedule where one receives the raise and the other does not.

Definition: Want an outcome which is the same as doing T1 first and then T2 or vica-versa. Serializiability.

Gold standard is to ensure serializiability for any commands, and any amount of parallelism.

Gold standard mechanism:

Divide the data base into granules (bits, bytes, records, …)

Lock everything you touch.

Ensure that locking is 2-phase, i.e. a growing phase and then a shrinking phase.

(In practice grow phase is whole xact, shrink phase is at commit.)

Deep theorem: 2 phase locking → serializiability

Easy generalization to Share (read) locks and exclusive (write) locks

Therefore, lock everything you touch, hold all locks to EOT.

Generally called two phase locking, or dynamic locking. Used by ALL major DBMSs.

Devil is in the details:

How big a granule to lock?   Records (page level locking gets hammered)
However, what to do with

select avg (salary)
from emp

don't want to set 10**7 locks.

Answer lock escalation to table level. If you set too many record locks in a table, then trade it in for a table lock.

What to do if you can't get a requested lock? Wait

What about deadlock? Can happen. Periodically look for a cycle in the "waits for" graph. Pick a victim (one who has done less work) and kill him. I.e. run crash recovery on his transaction (to be discussed next time).

Alternative: time-out and kill yourself.

Possible to starve. Repeated execution, retry cycle….

Doesn't happen in practice. In a well designed OLTP system, the average Xact does not wait. If not true, then redesign app to make it true. Rule of thumb: probability of waiting is .01 or less.

To avoid deadlock possibility:

All at once lock request (silly in practice)
Order all locks and request in order (silly)
Pre-emption (murder – not really done in practice)

What about auxiliary structures?

Lock table: must use latches, semaphores, etc. to serialize
Buffer pool: ditto
System catalogs (table table – drop table does a delete in the table table – other Xacts read a cached version -- generally finessed. Often the #blocks is in the table table. If updated, then do it with latches, not locks, …)
B-trees: too expensive to latch the whole tree

Hence, latch single blocks on access. Go to descendent block, latch it, release one up (called latch crabbing). Get to page to update, and latch it for the update.

Lehman& Yao have a scheme to avoid the latches (red book paper) – at the expense of noticeable complexity.

Halloween problem (urban myth that System R guys discovered this on Halloween). Aka phantom problem

```
T1:  begin xact
        update emp (set salary = 1.1 * salary)
        Where dept = 'shoe'
     End xact

T2:  begin xact
        Insert into emp values ('George', 20,000, 'shoe')
        Insert into emp values ('Hugh', 30,000, 'shoe')
     End xact
```

At beginning of xacts, there are 3 shoe dept employees: Bill, Sam, and Fred.

Suppose the emp table is sorted in storage on name, and holes are left for inserts. Suppose query plan for T1 does a scan of emp. Consider the following sequence of operations:

T1: update Bill
T1: update Sam
T2: insert George
T2: insert Hugh
T2 commits and releases all locks

T1: update Hugh(!!!!!)
T1: update Fred
T1 commits

Both xacts obey locking protocol – but result is not serializiable.

Issue is lock things you touch – but must guarantee non-existence of any new ones!!!

How: predicate locking (tried – doesn't work well – need a theorem prover – also, conflicts are data dependent)

How range locks in a B-tree index (assumes an index on dept). Otherwise, table lock on the data.

Escrow transactions

```
Begin Xact
        Update flights (set seats = seats -1) where flight = 234
        .
        .
        .
End xact
```

Locks 234 for the duration of the transaction. Nobody else can get a seat. Two transactions can go on in parallel as long as they perform only increment and decrement operations. Xacts commute, so everything is ok

However, if a Xact aborts, have to be careful with recovery logic. Forward pointer to Aries discussion.